CSE 586 – Distributed Systems
GitHub Notification System
Project 1: Phase 3

Documentation

Team number: 78

Siddharth Sankaran (50421657)

Shriram Ravi (50419944)

06 November 2021

1. Overview

This purpose of this document is to provide a detailed understanding of the publish-subscribe distributed system that has been implemented as part of Project-1 of the Distributed Systems course. This document will talk about the design, data flow, routing algorithm and brief overview of the steps to implement the code.

The system implemented here is a GitHub Notification System that allows a user to receive notifications from their preferred repositories. The user subscribes to different repositories available in GitHub by specifying the owner, repository, etc. in the front-end web application. The user then receives notifications of the latest commits in that repository.

The system also allows the publisher to send advertisements to the users through a simple user interface. Hence every time a user logs in, they can see the advertisements published by the publisher. The publisher can de-advertise to stop the publishing of the advertisement.

Users can freely subscribe and unsubscribe from repositories from time to time making the system dynamic and intuitive.

2. Design

The system is implemented with 4 components:

2.1. Publisher

The publishers in this implementation are the different repositories in GitHub that the user has subscribed to. Every commit to the subscribed repository will be published in the database and will trigger a notification to the user. The publisher end is implemented through a Python Flask application. The GitHub public API is called with specific arguments for owner, repo, time to receive the latest commit messages from the repo since the specified time. The publisher pushes this information (publish() function) to the middleware (broker network) which then inserts it into the database. The publisher is dockerized and run as a separate container that performs periodic API calls and pushes information to the middleware.

2.2. Subscriber

The subscribers in this implementation will log into the web application with a username and can subscribe to any public repository available in GitHub. This is enabled through a simple front end Flask application that allows the user to pick the repository they want and add a subscription. This pushes the subscriber information to the broker network (subscribe() function) which inserts it into the dataset.

Based on the list of subscriptions of a user, the front end receives appropriate notifications from the broker network periodically when new information is published. The front end has a notifications tab that allows the user to see the latest updates in each repository.

The front end also has tabs to add new subscriptions and unsubscribe from existing topics.

2.3. Middleware – Broker network

The middleware is implemented through a distributed approach with the help of a network of brokers. The broker network is that is responsible for routing the publish and subscribe requests through the systems. The network works with the help of a well-designed rendezvous routing algorithm that ensures minimal traffic and offers improved scalability, performance and fault tolerance. The routing algorithm is explained in detail further below.

In this implementation, the broker network is the intermediary that is responsible for enabling indirect communication between the publisher and the subscriber. The brokers are also the only entities in the system that can interact with the database. Accordingly, they are equipped to receive data from the front end for the following requests: new subscribe, new unsubscribe, new login, new advertise, new de-advertise. Similarly, they can also handle push requests from the publisher end containing the data of latest commits. The broker network is responsible for updating the database accordingly.

The middleware is also responsible for pushing notifications (notify()) to each user when the user is online. This is done with a simple online flag that is turned on when the user logs in. The system then performs matching to match the user's subscription list and the latest update seen by the user with the list of messages from the database to pick out the right set of unique notifications to each user. This notification is pushed to the frontend notifications tab that can be viewed on refreshing the webpage.

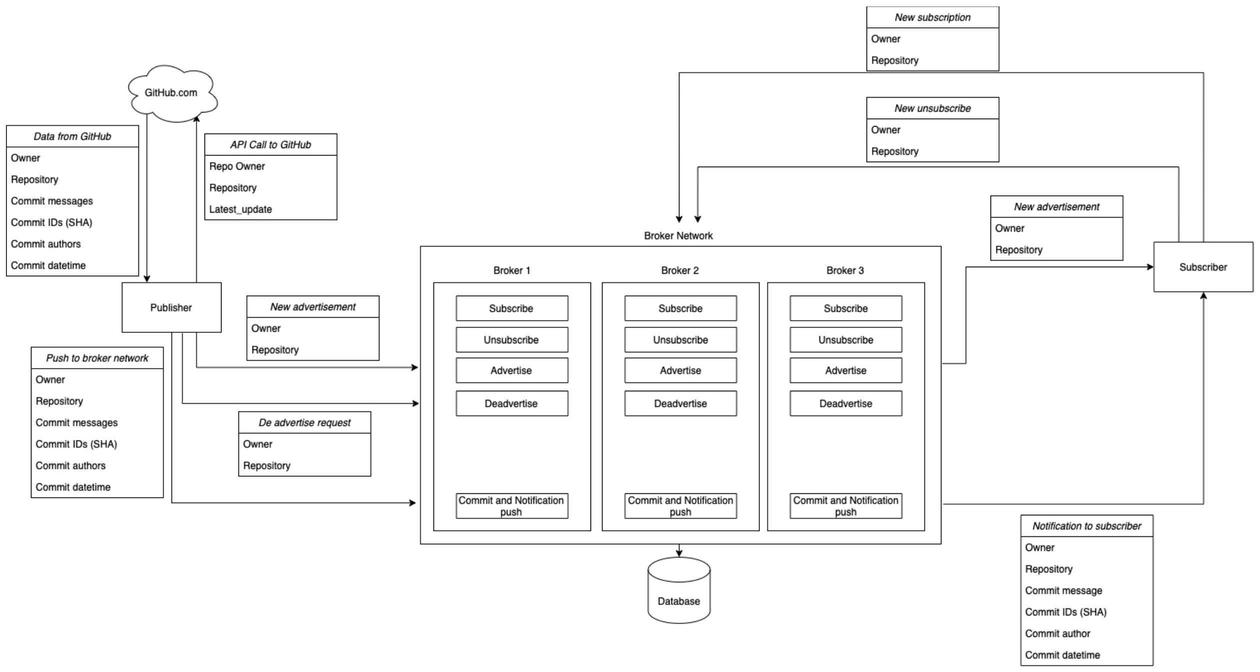
Each broker is containerized with a different hostname, IP and is exposed to a different virtual port. They are created from the same image but using different hostnames and IP addresses. In this way it is relatively simpler to increase the number of brokers in situations where a high load is expected in the system.

2.4. Database

The database used in this system is MongoDB. Each database is organized into collections and documents. The database created for this example is subscribers_db which is composed of 3 collections

a. subscribers_db – Contains list of subscribers with their subscription list. Each subscription list contains the list of repos and the latest update seen by the user
b. topics_db – Unique list of topics that all users are subscribed to. Also contains the latest update of each topic that is present in the database
c. commit_messages_db – Set of commit messages for each user subscribed topic

3. Dataflow Diagram

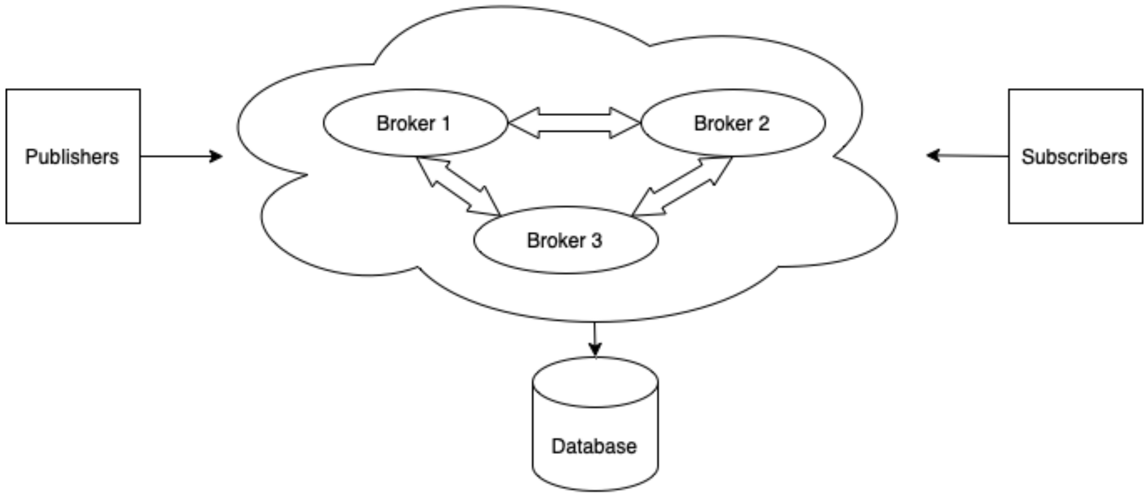


The above diagram depicts all the data transactions in this project.
On the left-hand side of the diagram is the publisher end. Here the publisher flask application polls the GitHub public API to obtain the recent messages from the all the subscribed topics. This returns the message and message related information like id, timestamp, author, etc. to the publisher. The publisher passes this on to the broker network through a POST http request to the broker network, specifically to the commit_notif_push() method. The request is posted to a random broker (i.e. in reality this could be the equivalent of pushing to the nearest router). This publish method is refreshed every minute to check for new updates and messages and promptly pushes these to the broker network periodically.
The broker network receives the messages through one of the brokers and routes them further into the network towards the direction of the broker that is responsible for handling that specific request. The final broker receives the new messages and inserts them into the MongoDB using the insert_one() method from the PyMongo module. Then this method also begins performing the matching operation every time a new data is pushed to the data base. The matching operation interacts with the database to find the list of subscribers online, fetch their respective subscriptions and filter the latest messages. The notification for each user is

filtered based on the last message seen by the user for each of his subscription. This notification is then pushed to the frontend with a POST request to the notifications page. Once the notification is sent each user’s latest update timestamp is updated with the time of the latest commit. This is done so that each user does not receive outdated notifications again and again. Each topic’s latest commit time is also updated in the database. This is done so that the publisher doesn’t publish information on topics that are already present in the database
In the right-hand side the subscriber inputs data in the form of new subscription, specifying owner name and repository. The subscribe() function takes this information and passes it onto the one of the brokers from the broker network. The network receives the message and efficiently routes them further into the network towards the direction of the broker that is responsible for handling that specific request. The final broker receives the new messages and inserts them into the MongoDB using the insert_one() method. The routing in between brokers is handled by the rendezvous routing algorithm that is explained further below.
For creating an advertisement, the publisher can visit the ‘/advertise’ route in the web application to create a new advertisement. This info is sent to the one of the brokers in the network which modifies the advertise flag in the database to 1, enabling a rolling advert in each webpage. De-advertise is performed by moving to the de-advertise tab in this page. This will send a POST request to one of the brokers, which interacts with the database to change the advertise flag to 0, thus removing the rolling advertisement from each page.

4. Routing Algorithm

The system of brokers utilizes a rendezvous routing algorithm. In this case, we have implemented the distributed network with 3 brokers each of which, are connected with each other as shown in the broker network diagram below:
The network can be scaled to n brokers as per need, but for the current implementation we have demonstrated with 3 brokers. To add more brokers, we can reuse the existing broker code to initialize a new container with a new hostname.



Broker Responsibility:

In this network each broker is responsible for a set of topics. In order to keep the topic to broker mapping dynamic, we have assigned topics to brokers based on the starting alphabet of the topic (name of the repository in this case). So, the broker responsibility looks like below:

Broker	Topics
Broker 1	All the repositories with starting alphabet between A and H
Broker 2	All the repositories with starting alphabet between I and Q
Broker 3	All the repositories with starting alphabet between R and Z

This was done to make sure the load is evenly balanced between all brokers and to ensure that the network doesn't suffer in times of high traffic.

Routing:

When a broker receives a publish request, the broker calls the EN() function to identify the broker node responsible for that particular topic. A check is performed to check whether the current broker is part of the list of brokers that are responsible for this topic (rvlist). If the current broker is not responsible is not in charge of this topic, it forwards it to all its neighbors. In case the current broker is responsible for this topic, a match operation is performed to filter the required notifications as per the subscription list. Then the filtered notifications are sent as notifications to the subscribers who are currently online.

The list of neighbors who are connected to the broker and the list of subscriptions for which the broker is in charge, are stored as global variables in each broker's container.

When a broker received a subscribe request, the SN() function is called to find the broker which is responsible for that event. This returns a list of broker names. The current broker checks if it is present as part of this list, and if not, it forwards it to all its neighbors. When it is present in the list, it adds the subscription to the subscription list and to the pushes to the database as well.

Using this algorithm, we can ensure that the middleware is not overburdened with too many requests. By assigning broker responsibilities and designing this routing algorithm we can ensure reduced load on each broker and hence improved performance and scalability. Since there are multiple brokers, the system can easily handle the failure or one or more brokers as well. This approach can be further improved by adding more brokers and making the routing algorithm more dynamic.

5. Technology Stack

For this project we have used a combination of html, CSS, python (Flask), MongoDB for implementation

a. Frontend – HTML, CSS, Python (Flask)
b. Backend – Python Flask
c. Middleware broker network – Python, MongoDB

6. Team Contribution

Name	Person Number	Contribution
Shriram Ravi	50419944	a. Login b. Unsubscribe c. Advertise d. Notification handling e. Rendezvous algorithm – Subscription handling
Siddharth Sankaran	50421657	a. Logout b. De-advertise c. Subscribe d. Publishing - API call e. Rendezvous algorithm – Publishing handling

7. **Steps to run the code**

Please find below the steps to be followed to obtain a successful implementation of Phase 3 of the project

1) In Terminal, navigate to frontend folder and run to create the frontend docker image
- `docker build -t frontend-image .`

2) Use the image to create the front-end container
- `docker run --name frontend-container -p 5003:5003 frontend-image`
- You can create multiple instances of front-end containers (Subscribers). These needs to be exposed to different local ports.

3) This will run the frontend container on port 5003 in localhost

4) In another terminal, navigate to the api_call folder and run the following command
- `docker build -t apicall-image .`
- `docker run --name apicall-container -p 5002:5002 apicall-image`
- This will create a container designed for fetching publisher data from the GitHub

5) In another terminal, navigate to the backend folder and run the following command
- `docker-compose up`
- This will create separate container for database (MongoDB) and initiate it. This container will be placed under a docker network.

6) In another terminal, navigate to the backend folder and run the following command
- `docker build -t backend-image .`
- `docker run --name backend_broker_1 -h backend_broker_1 -p 5101:5101 backend-image`
- `docker run --name backend_broker_2 -h backend_broker_2 -p 5102:5101 backend-image`

- `docker run --name backend_broker_3 -h backend_broker_3 -p 5103:5101 backend-image`
- This will create three separate flask containers which forms the rendezvous broker network and initiate them.

7) In another terminal, run the following command to add the front-end container and the api_call container to the above docker network so that all three containers are in the same network
- `docker network connect backend_default frontend-container`
- Do the same for all the subscriber containers to connect them to the docker network

8) Connect all the broker containers to the docker network
- `docker network connect backend_default backend_broker_1`
- `docker network connect backend_default backend_broker_2`
- `docker network connect backend_default backend_broker_3`

9) Connect the API Data provider to the same docker network
- `docker network connect backend_default apicall-container`

10) To verify if all three containers are in the same network run,
- `docker network inspect backend_default`

11) Open a browser and go to
- `localhost:5002`
- This will start the data fetcher container and update the appropriate GitHub events to the DB

12) Open a browser and go to
- `localhost:5003`

- Provide appropriate inputs. These inputs will be added to the MongoDB once the submit button is clicked
- This is extensible to all the containers created with the frontend-image